Application of Functional Programming in the Energy Industry: A Local Energy Market Simulator Use Case

Amine Zouhair
EDF R&D - Dep MIRE
Palaiseau, France

Nadine Kabbara
EDF R&D - Dep PERICLES
Palaiseau, France

Olivier Boudeville
EDF R&D - Dep PERICLES
Palaiseau, France

Florian Mancel
EDF R&D - Dep MIRE
Palaiseau, France

ABSTRACT

This article presents the efforts made to develop a simulator for local energy exchanges by means of an energy marketplace designed as a multi-agent model.

The objective of this article is not to focus on EDF's industrial use case by itself, but to share elements of experience regarding our use of functional programming in order to create this specific simulator and the generic layers on which it is built.

CCS CONCEPTS

• **Applied computing** → *Engineering*; • **Computing methodologies** → **Parallel programming languages**; **Modeling methodologies**; **Distributed simulation**; **Discrete-event simulation**; **Agent / discrete models**; **Concurrent programming languages**.

KEYWORDS

Functional Programming, Agent-Based Simulation, Meta-programming,, Distributed Systems, Energy Markets.

ACM Reference Format:
Amine Zouhair, Nadine Kabbara, Olivier Boudeville, and Florian Mancel. 2021. Application of Functional Programming in the Energy Industry: A Local Energy Market Simulator Use Case. In *Proceedings of IFL21*. ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Some context is of use to better understand the constraints that apply whenever considering the use of Functional Programming (from now shortened as FP) from an industrial point of view like the one in which ACME[1] was developed.

EDF[5], the supporter of this study, is a large-scale multinational energy utility operating on the three main associated industrial perimeters of electricity: production (with notably 56 nuclear reactors in France, but also hydropower, marine energies, wind power, solar energy, biomass, geothermal energy and fossil-fired energy), transport (to convey larger quantities of electricity across a country) and distribution (so that all customers, including residential ones, can be connected to the resulting electrical grid).

EDF maintains a strong effort of in-house R&D, operating in varied scientific fields, ranging from power electronics and industrial control, economics of energy, neutronics to urban planning, numerical analysis, structural mechanics and industrial risk assessment.

The many operational requirements of a larger utility incur significant needs in terms of software development. Such a sustained activity led to best tackle the corresponding software challenges by creating transverse R&D divisions whose role is to bridge the gap between mainstream industrial software engineering and applied research, covering a wide range of topics often close to computer science: from HPC (*High Performance Computing*) to the architecture of larger information systems, from artificial intelligence to distributed systems, from applied mathematics to virtual reality, cybersecurity, etc.

The corresponding tools developed for the industry's projects fall into various categories, from one-shot proofs of feasibility to generic, feature-rich platforms, some of them being released to the public as open source software [31].

Even though the vast majority of EDF R&D's larger software projects have been relying on programming languages that are mostly imperative (often Fortran, then C, C++, Python), some level of interest in functional programming still arose. This could be seen for example through some sessions of internal training addressed to developers of HPC solvers, in the belief that some exposure to FP could improve even one's use of imperative languages regarding clarity, correctness (ex: with a more direct link to formal proof) if not efficiency (ex: to support larger linear operations).

This ongoing interest in FP could also be seen through the involvement in European projects like RELEASE[28] or in the organization in 2012 of a CEA-EDF-INRIA summer school on *Functional Programming for Parallel and Concurrent Applications*[15].

Another concrete element regarding EDF's use of FP lies in the ACME simulator discussed in this article, whose purpose, industrial context and requirements will be presented in the next section.

ACME, as a direct application of the Sim-Diasca simulation engine, is by design fully implemented as a functional program, and this specificity will be discussed in-depth through section 5.

[1] ACME stands for *Autoconsommation Collective et Mobilité Electrique*, French for *Collective Self-Consumption and Electric Mobility*.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
IFL21, Sep. 2021, Radboud
© 2021 Association for Computing Machinery.
ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

2 A LOCAL ENERGY MARKET SIMULATOR

Electrical systems are currently facing vast emerging challenges that are revolutionizing their development into smarter power grids. These challenges mainly originate from the increase in decentralized energy production (renewable energy sources (RES), storage batteries) and the rise in popularity of electric vehicles (EV) - both of which stemming from the need of better addressing the ongoing climate change.

The abundance of an increasingly decentralized production of electricity made the traditional centralized design of the power system obsolete, and transformed a mostly static, hierarchical grid into an active network that allows for bidirectional flows of energy and information between customers and electricity suppliers. This change leads to the emergence of 'prosumer'-type behaviors, where energy actors become both producers and consumers, that actively participate in the electricity supply system and its balance.

In order to adapt to such changes, incentive mechanisms can be devised at a local level, so that a mixed community of consumers and prosumers can better synchronize their electricity consumption with local production. 'Prosumers' can profit from selling their production surplus or energy stored in their batteries at a price higher than offered by the regulated mandatory feed-in tariffs. Similarly, consumers offer to shift their consumption (mainly controllable uses such as washing machine, water heater, electric vehicle charging, etc.) to benefit from locally-produced renewable electricity at profitable prices.

Such mechanisms can be provided by means of a Local Energy Market (LEM) [32],[27],[26],[23]. A LEM allows prosumers and pure consumers, connected to the same area of the distribution network, to virtually trade electricity by means of interacting with an energy marketplace platform.

The promise of these newer organizations in favor of electrical self-consumption are numerous: support of RES/EV integration, better local energy management, decrease in the need for expensive grid expansion and in average energy costs.

Previous works on LEM can be classified according to three main intents: studying their potential and benefits depending on different market designs, assessing their utilization for local energy management, and using blockchain technology with smart contracts for their real-life implementation.

Marketplace trading is typically addressed by participating in a multi-unit double auction system that matches supplies and demands on a day-ahead or intra-day basis. In [16], a novel market design respecting grid constraints was proposed. In [23], a market allowing P2P trading with battery flexibility was tested. Two designs with batteries located at a centralized level and user level were considered. In [27], a comparison is conducted between different market designs with both zero-intelligent and intelligent strategies based on an agent-based simulation.

Regarding local energy management, in [24],[22], an intelligent management system based on a market mechanism is proposed. These results validated the benefits of such new management in reducing peak loads while increasing one's autonomy from the grid.

A first proof of concept for a blockchain-based energy market was developed in [25], and in [26] the eligibility of blockchain technology for operating decentralized energy markets was established. However, the socio-economic impacts of these markets and the technological evaluation and limitations of blockchain technology for real-life operation (in terms of scalability, robustness, transactions costs) was not investigated. In [32], which inspired some aspects of ACME, a blockchain-based platform and business models for LEM were designed and demonstrated. The technical and regulatory aspects of the simulator were assessed, and a smart contract library for energy applications was developed.

With regard to previous works, we noticed that battery storage technologies and EVs are often neglected when studying LEM, and that the agent-based models are rarely transformed into reusable simulators allowing for the exploration of model parameters. Also, most agent-based models of LEM utilize simulation engines of limited scalability, so the number of market participants in the simulations is often restricted due to computational limitations. Finally, a global simulator assessing both the properties of the electrical system and of the information system for local energy trades is rarely targeted. As a consequence, the ACME project aimed at developing a multi-usage simulator for local energy exchanges through a LEM that considers the additional dimensions of battery flexibility, consumption of electrical vehicles, and of the information system governing the associated exchanges.

The goal of our study is to evaluate the techno-economic interests of local energy exchanges, and to help understand the various factors that may impact these exchanges and the operation of the market, like: the number of participants, the proportion of prosumers/consumers, the capacities of the solar batteries, and the type of information system (centralized server versus blockchain-based infrastructure) that organizes such exchanges.

These factors correspond to as many inputs of the ACME simulator, which in turn evaluates the following metrics: the percentage of energy that ends up being self-consumed in the community, the average local electricity prices, the average electricity bill of a participant, the profile of consumption peaks, and the proportion of unfulfilled bids and asks addressed to the marketplace.

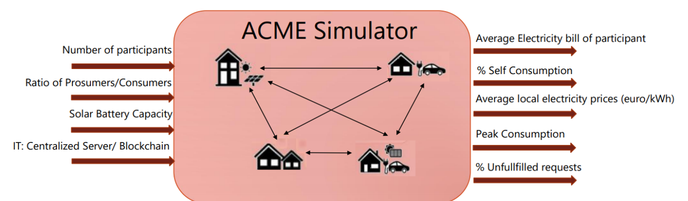

Figure 1: Inputs and Outputs of the ACME Simulator

3 A STUDY IN TERMS OF COMPLEX SYSTEMS

3.1 Introducing the Field of Complex Systems

Complex systems are systems composed of often numerous components that are at least partly autonomous yet are bound to interact, each according to its own purpose, patterns, features, state properties and constraints.

Quite often, these systems tend to not be deterministic, with at least some of their components exhibiting partly stochastic behaviors.

For such systems, the whole is more than the sum of its parts, in the sense that the collective behavior at a global level can hardly be inferred directly from the component level, and that in the general case no macroscopic description of it can be done in terms of equations; more generally most of these systems are believed to be irreducible to any aggregated modelling. Their emergent system behavior is intrinsically dependent on the actual interactions between their parts or between these systems and their environment, often with interdependent bonds and feedback loops.

Since the 1970s, many examples of complex systems have been found in biology, transportation, social networks, urban planning.

Generally a wide range of approaches can be applied to study systems of all kinds, the least demanding ones being expert assessments, quick computations based on orders of magnitude, or more detailed spreadsheet-based evaluations. However, at least for some metrics - notably regarding performances and scalability - such lightweight measures can hardly be applied to most complex systems, as these measures fail to account for their temporal characteristics (ex: for electricity like for communication networks, average energy flows or bandwidth do not tell much about the peak requirements, whereas in many cases the real driver of the design of these system is actually these extreme values).

Finally, in the general case, neither comparisons against validated, reference tools can be done (they seldom exist) nor reduced-size actual experiments are applicable (short of leading to results that could be extrapolated further).

As a result, most complex systems can only be addressed thanks to (computer-based) simulations - hence the purpose of the ACME simulator. However such a reasoning is relevant only if the system of interest for ACME is a complex one indeed - which the next section will attempt to establish.

3.2 Decentralized Electrical Systems as Complex Systems

Larger organisations of all sorts are likely to exhibit traits pertaining to complex systems whenever they involve rather autonomous peers interacting according to rich patterns in dynamic, heterogeneous environments.

This becomes quickly the case for many distributed applications whose potentially unreliable nodes have to collaborate in spite of non-ideal, latency-ridden networks; for them no global state or clock are available, leading to hard impossibilities like the ones established by the CAP theorem[1].

An electricity utility has to deal with such distributed systems. This includes smart power grids, federating large populations of meters (ex: 35 millions in France) and concentrators (700,000 of them) interacting through PLC communication in order to implement distribution-related services in spite of various classes of failures and technical issues in terms of repeating and crossover; smart grids are certainly among the largest artificial distributed systems, and lead to unprecedented challenges in terms of reliability, performance, scalability, security, cost efficiency and maintainability.

For long, electrical systems in general and electrical grids in particular have been mainly modelled according to a mathematical decomposition following a top-down hierarchy that corresponds to their statically ramified architecture [17]. This kind of approach is relevant to monolithic models where computable functions can be derived and studied.

To satisfy the requirements expected from the new generation of power grids, there is a need for bidirectional, real-time communication networks for data collection and processing[20]. The modern grids should be able to collect all kinds of information regarding electricity consumption from smart meters and production profiles (from centralized or distributed sources) in order to increase their efficiency and stability.

When describing the local electric community where the exchanges are to take place, we find many characteristics that point toward it being a complex system:

- System Decentralization: the system is composed of many actors (producer/consumers), distributed at multiple levels, and able to interact and develop collective behaviors that impact the overall electrical system stability; of course, should the IT system be a blockchain, the centralisation and level of coordination of these systems are further decreased.
- Heterogeneity of elements: the system is composed of energy consumers who take decisions on their consumption and production if any, energy markets, an IT infrastructure, an electrical infrastructure, electricity suppliers, ... Each of these stakeholders, in the pursuit of its individual interest, acts autonomously and based on partial knowledge.
- Information Data: sensors and actuators that are spread all over the system enable stakeholders to take varied, dynamic actions, possibly in a programmed way

4 ACME STUDY : A THREE STEP APPROACH

The ACME virtual experiments are conducted according to a more general process found relevant for the study of most complex systems:

1. Modelling the system of interest in the light of the metrics of interest
2. Translating these models into elements able to be technically evaluated by a relevant simulation engine
3. Exploiting the resulting simulator in order to generate new domain-specific knowledge

4.1 Modelling the System of Interest

One claim of this article is that the success of a project in terms of complex system simulation depends heavily on two characteristics:

- how the implementation of models derives naturally from their domain-level specifications
- to which extent these models can be sheltered from the complexity of the evaluation runtime, notably regarding their implementation language and their integration to the underlying simulation engine

Through the ACME example detailed here, we aim to showcase that FP can offer solutions that have been validated through experience and that are appropriate to secure both characteristics.

For domain experts (ex: project managers in charge of the roll-out of automated metering systems, urban planners or, closer to ACME, designers of smart charging infrastructures for electric vehicles) and also for most developers on the market, the natural way of specifying models is often to express them based on simple abstractions and according to schemes where usually an imperative, object-oriented approach applies.

Indeed, more often than not, the modelling activity starts with actual, concrete instances (ex: a given smart meter involved in a user story), generalizes them by merging their involvement into the various scenarios established, and splits actors into a set of roles.

This allows abstracting these elements into relevant blueprints (ex: introducing a meter class), then separating their state (based on attributes, which quickly need to be typed) from their behavior (soon to become methods), and factoring common elements through inheritance (ex: both a smart meter and a concentrator being then fruitfully captured as a special case of a then-introduced abstract communicating device).

Studying a complex system incurs some challenges. An analytical approach, whether through differential equations or stochastic processes, is almost impossible for the level of complexity that arises when tackling the intrinsic properties and the interactions of IT infrastructures, electrotechnical systems and, more importantly, human decision processes. We are required to adopt a systemic approach in which the behavior of the system components and their interactions are first tackled on their own, before the emergence of global properties happens.

Since we are in front of a system with a high number of intervening variables, traditional systemic models like Rule Based Systems are not suited for the task. The approach followed for ACME stems instead from the constructivist models [10] (Individual-based Models, Multi-Agent Systems, etc.).

When dealing with a complex system of systems like an energy community through a constructivist approach, we consider multiple types of actors with different (sometimes conflicting) goals and knowledge. The Multi-agent Systems approach is particularly well-suited for this kind of problem. Indeed, by choosing to consider each agent individually, we can concentrate on what this agent knows about the world without risks of awareness violation, use the domain knowledge available from system experts to design a behavior as close to reality as possible, while using the inherent modularity allowed by this approach to adapt our model to the evolving needs of the stakeholders - moreover rather inexpensively once it has been first implemented.

The ACME model covers the three essential domains required for a proper operation of the LEM: a simplified electrical system, an energy marketplace model [21], and a software system model.

Within each domain, the major actors accounting for the simplified system have been modelled as active, stateful autonomous agents, each in charge of a set of roles.

In the electrical system layer, the main actors that we identified are: household supervisors (driving EMS, for *Energy Management system*), electric meters, electrical appliances (such as washing machines or dishwashers), solar batteries, solar panels, electric vehicles, charging spots and electricity suppliers.

Regarding the energy marketplace, since we wanted to explore the impact of different IT infrastructures on the usefulness of these trading organizations, we chose to explore two main types of backends. The reference one corresponds to the use of a classical, centralized IT system hosted by an application server that would be operated by a dedicated economic actor. Due to the simplicity of the corresponding architecture and to its small scale (by design a local community may regroup up to a few hundred participants), we do not see such a system as a complex one.

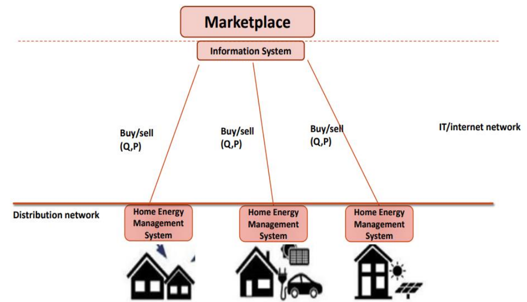

Figure 2: The LEM layers covered in ACME [23]

However a goal of ACME is to investigate the upcoming architectures that could support LEM use cases, namely Distributed Ledger Technologies (DLTs). As a result, a second IT option is considered in our simulations, in which the targeted marketplace is implemented based on smart contracts executed by a blockchain.

The blockchain network considered in our model is a permissioned consortium where properties such as privacy and security, energy consumption (choice of consensus protocol), and scalability and reliability are to be evaluated.

This second IT option, meant to be compared to the reference one, may be especially relevant in the context studied by ACME: not only prospects of drastically reducing the energy consumption of blockchains are real (when switching from a proof of work to, for example, proof of stake), but also a community having invested on decentralised means of electricity production may value the increased autonomy towards the distribution network - and then be reluctant to be centralised again because of the IT infrastructure.

- For centralized IT, the classes needed to model the most important parts of this application (mainly the risks of single point of failure and latency) are: Server for dealing with requests, Centralized Database and the Marketplace Organizer Service.
- For blockchain-based transactions, the design was based on smart contracts built on a minimalist blockchain model[2]; the classes needed for this model are: Participant Client (inheriting from DLT Client), Participant and Marketplace Organizer Contract (inheriting from abstract Smart Contract), Simplified Blockchain Model (a light node representation with transaction pools, block creation based on inter-block target durations and capacity), and the blockchain ledger itself

Our models are described in UML 2.0 (*Unified Modelling Language*), which offers a standardized way to design systems. The

[2]Notably because the underlying P2P network is abstracted out and the blockchain model focuses solely on its functional service.

UML representation is especially useful in industrial projects that most often follow widely standardized protocols and tools. Similar to an object-oriented approach, the multi-agent model can be visualized using class diagrams, activity diagrams and use case diagrams[11]. We made use of these diagram representations to specify the multi-agent models describing the functioning of the local energy market and the relationships between its different actors (class diagrams), and also to validate the outcome with domain experts. The UML use case and activity diagrams are used to represent the perceptions and actions of each agent.

4.2 Deriving from the Models a Legit Simulation

A simulation is the imitation of the operation of a target process or system over time. When dealing with complex systems where it is nearly impossible to derive computable functions for the metrics of interest, simulation is often the unique way to proceed.

The multi-agent approach adopted in the modelling stage grants us some freedom when it comes to implementing the simulator:

- The inherent modularity leads to simpler programming when compared with developing a monolithic, centralized agent - especially in an interactive cycle of simulation development, where the final users of the tool can come with new propositions about the sophistication of an agent behavior and its interactions.
- Concurrent evaluation is also a possible outcome of this choice since every agent's behavior at a given time is a function of what has been encapsulated prior, so the computation of the system's behavior have chances to be massively parallelized (each agent on its own) at the price of regularizing the time of the simulation, as discussed in section 4.2

We will also see in section 5.2.1 that the multi-agent paradigm finds in Erlang a direct, idiomatic implementation, and, in section 5.6, that a corresponding simulation finds in Sim-Diasca a suitable runtime.

However, the UML 2.0 representation of the multi-agent system, based on elements with adequate states and methods, suggests that an object-oriented programming approach may be the most idiomatic way to translate the models into running code.

A potential discrepancy exists there, as Erlang was chosen for computational reasons discussed in next sections, yet its paradigm is purely functional and not specifically object-oriented. To get the best part of both worlds and avoid a mismatch with the practices of domain experts and modellers, there is a need for a suitable translation from OOP concepts to the purely functional ones offered by Erlang. Then all ACME models could derive, directly or not, from an abstract mother class that the simulation framework would provide, and agents like electricity consumers and marketplace participants could share part of their specified states and behaviors, while still being able to be further specialized on a per-type basis, through subclassing and the overriding of some of their domain-specific methods. The technical consequences of these requirements will be discussed in 5.4.

Figure 3 illustrates a simulation with fixed consumption and production profiles; the corresponding evolution of the local energy prices found based on the effective trades on the LEM and using the multi-unit double auction mechanism is presented. Following these prices, the overall electricity bill of each participant can be computed by aggregating their consumption satisfied from the local market and from their supplier.

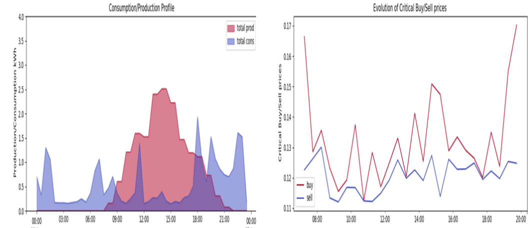

Figure 3: The evolution of local energy prices in the community according to aggregated consumption/production demands in a day during the simulation

In figure 4, the distribution of the consumption throughout the day according to different origins (local production, battery, supplier) is shown. This allows finding the percentage of self-consumed energy in the community, and to assess the peak load reduction of the distribution grid.

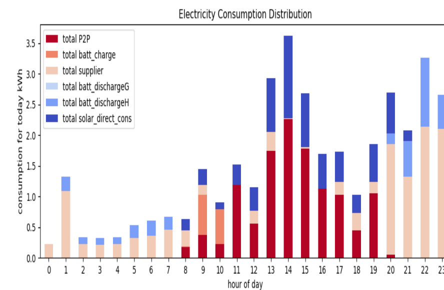

Figure 4: The distribution of energy consumed in the local community by different origins in the simulation

Still considering ACME only as a black-box user of the underlying simulation framework detailed below, we could observe that the computation time of the simulations was scaling well with the number of actors in the simulation, especially when we consider that, since an actor has the potential to interact with nearly all the others, the number of messages and interactions grows exponentially with the size of the simulated universe.

4.3 Exploiting the Simulation to Generate New Knowledge

The simulator, once developed, can provide its users with insight on different areas of interest. One such example is the economical benefit of a participant in this LEM, the distribution of energy in this community, the influence of the information system on the market's operation, the breakdown of the energy consumed by a household or the outcome of the solar energy captured. Here below are some graphs that were generated by the simulator for use cases specified by the domain-minded users.

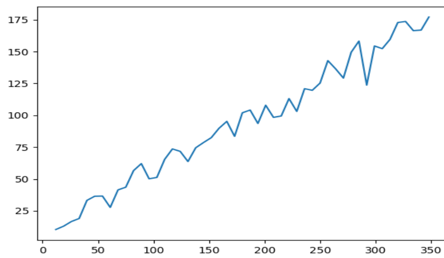

Figure 5: The computation time of the simulation running for a year vs the number of agents in the simulation

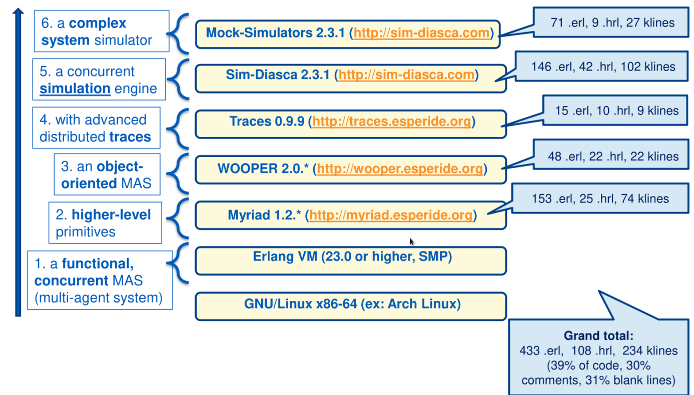

Figure 6: ACME Software Stack

The simulator can therefore be depicted from a black-box point of view (cf. Figure 1): for certain inputs describing a universe (number of suppliers, consumers, prosumers, solar irradiation,...), it will unwind the interactions between the different agents and output metrics of value for the user (percentage of renewable energy consumed during the simulated time, the average bill of a participant community, etc.). Such an exploration is particularly useful in the study of complex systems where emergent behaviors are to be observed based on different scenarios and actors decisions.

However the use case at hand goes beyond simply exploring the model through the experienced eye of field experts: the current work on ACME aims to explore its model based on evolutionary metaheuristics and automate the search for optimal outcomes.

OpenMOLE (Open ModeL Experiment)[6], which is an open source modelling exploration software, can serve as a tool to perform this model exploitation, allowing to gain new knowledge about complex models by designing virtual experiments through parameters exploration.

The OpenMole workflow engine offers parallel execution environments for naturally parallel processes, which makes it a particularly suitable software layer to deploy on the EDF cluster for further analysis of the model though more elaborate sensitivity analysis and calibration. This is particularly important in models with stochastic components like the ACME model.

The connection of the ACME simulator to OpenMOLE has been done by containerizing the ACME simulator application with Docker and coding an adapter layer in Python. OpenMole will use Singularity to run the Docker images and interact with the model, launching as many simulations as needed for the targeted exploitation use cases.

5 ACME WITH REGARD TO FUNCTIONAL PROGRAMMING

5.1 A Glimpse at the ACME Software Stack

Exactly like the mock simulators that are provided as examples with the public, open-source distribution of Sim-Diasca[29], the ACME simulator sits at the top of the software layers pictured in figure 6.

Such a layered approach has been designed in order to favor separation of concerns, and to gradually specialize a general-purpose (functional) programming environment into, ultimately, a simulator able to evaluate any complex system of interest.

The next bottom-up walk-through of these layers will explain their respective nature and purpose.

5.2 Runtime Layer

5.2.1 Erlang/OTP Overview. The full stack of ACME is based on the Erlang/OTP language[19], a concurrent, functional programming language that was first introduced in 1986 by Joe Armstrong, Robert Virding and Mike Williams. Erlang has been developed since then within Ericsson, and starting from 1998 has been released as free and open-source software.

Erlang is a functional programming language, based on immutable data (single assignment), and eager evaluation. It relies on recursion and pattern matching, and provides the staples of most FP languages, such as lambda-functions and higher-order ones, pure guards, closures, and tail-call optimizations.

Expressions and patterns are evaluated based on terms directly composed out of algebraic data types, knowing that a design decision of the language was not to provide any reference semantics regarding terms.

Besides its FP nature, a central element of Erlang lies in its concurrent mode of operation: every Erlang program consists of a set of logical processes that interact solely through asynchronous message passing (hence with a strict isolation enforced, and no shared memory). Each Erlang process is identified by its PID (*Process Identifier*), and sending a message to a process is as simple as sending an Erlang term to the PID designating that process.

These logical processes[3] are designed to be as lightweight as possible, with a very small memory footprint (hundreds of thousands of them can be evaluated by any inexpensive computer) and to be executed concurrently by a virtual machine, based on strong, dynamic typing[4] and a garbage-collected runtime system.

An Erlang node exists on a given host (computer), and thanks to its SMP schedulers is able to use all cores of all local CPUs in

[3]Erlang relies on green threading, typically implemented on top of multicore (SMP) architectures; Erlang processes are therefore mostly unrelated to the processes of the operating system.
[4]Erlang supports additionally type specifications[18], and includes built in tools in order to perform static type checking[4].

order to execute efficiently and in parallel all the processes that have been (based on an explicit placement) spawned on it.

Furthermore Erlang nodes are designed to be easily interconnected within a network. As inter-process communication relies only on opaque PIDs of processes that may indiscriminately either belong to the local node or to any remote one, by design Erlang programs can be seamlessly distributed. The language provides additionally the various concepts and primitives (such as monitors) that are necessary to properly handle distribution.

The net result is a FP language especially suitable to the implementation of concurrent programs, which was the main properties that we sought in order to develop the Sim-Diasca simulation engine presented in section 5.6.

5.2.2 Use of Erlang/OTP as seen from ACME. The ACME simulator, being powered by Sim-Diasca, corresponds to a rather unusual application of Erlang: whereas this language shines for distributed, fault-tolerant, soft real-time applications for which high availability is key, these features may not seem so much in-line with the requirements of use cases pertaining to the simulation of complex systems discussed in section 5.6.

However, to anticipate a bit on this section, a first point is that these simulations may either be fully standalone (being then able to be just evaluated as fast as possible), or may have to run on par with an external clock.

Indeed, in the former case (to which ACME belongs), the simulation is able to recreate a *full* virtual experiment, in which both the target system (in ACME: a local energy marketplace and its residential participants) and its context (here: the information system implementing these services, the electric vehicles, the household appliances, the suppliers, etc.) have been translated to models on which the simulation engine has full control; as a consequence it can schedule their evaluation as fast as the hardware resources permit.

On the contrary, the latter case is closer to emulation insofar that it involves an external clock source (generally deriving from wallclock time, possibly with a scale factor) that rules a part of the system of interest (ex: an actual smart meter device whose compliance shall be tested). The simulation can then be used to recreate the rest of the system and of its context, for example to emulate all the elements with which said smart meter is to interact, for a controlled testing thereof. For such a use case, provided that the simulation is fast enough (hence the interest in parallelism), it can outpace the external clock and thus be automatically slowed down by the time manager of the simulation engine in order to be just on par with that clock; in such a setting the soft real-time capabilities of the Erlang VM (notably its fair, non-blocking process scheduling and its concurrent tracing garbage collector) are of great interest.

Other traits of the Erlang language, concurrency and scalability, are even more essential for our use cases but will be discussed in layers above in the software stack.

Finally, some other language-level features of Erlang found no echo in simulators like ACME, notably the support of systems that shall never stop (thanks to hot code update) or for fault-tolerance: simulations are meant to be one-shot runs, and shall crash as early and as fully as possible whenever any model-level error condition is met, rather than trying to maintain any sort of service.

In practice, for the development of ACME, this translated mostly to sticking to pure Erlang, and not specifically relying on its associated OTP framework[5].

5.2.3 Hardware and System-level Considerations. Section 4.2 discussed why a major challenge of simulations of complex systems is to maximize the concurrency of their evaluation. How effective FP may be in order to devise proper parallel and distributed algorithms, these computations have to ultimately map onto actual hardware resources that are relevant in an industrial context.

For that we experimented various technical options potentially effective in order to run Erlang code.

One promising architecture for this use - long predating ACME - was the manycore cards, offering numerous independent processing units corresponding to a concurrent alternative to parallel, vectorized architectures such as GPU-based stream processors.

Experiments were thus made with a Tilera card, the TILEPro64[9], a cache-coherent mesh network of 64 VLIW ISA "Tile processors". The Erlang VM could be adequately cross-compiled to the TILEPro64 RISC architecture, and - almost transparently thanks to the Erlang toolchain and bytecode-based runtime - Sim-Diasca as well.

While this key step succeeded, the overall process of transferring code and data back and forth to the card was a bit cumbersome (knowing that extra tools, such as post-processing ones, were not as portable as the simulation itself) and, more importantly, making an efficient use of these cards would have required to adapt at least the load-balancing of the engine in order to take advantage of the underlying connectivity patterns of the tiles. Since such ad hoc developments exceeded the potential gains that we foresaw for our R&D use cases of that time (knowing moreover that this effort would have been specific to a rather non-standard hardware, of uncertain longevity), no further step was made in this direction; this test nevertheless confirmed a promising match between manycore architectures and at least this kind of Erlang applications.

Another parallel execution platform that was experimented - this time in the context of the RELEASE European project[28] - was the potential use of Erlang on supercomputers, platforms on which FP is probably less present. Indeed, in addition to the HPC clusters that will be mentioned next, EDF R&D used to rely on IBM Bluegene supercomputers, first the Bluegene/P generation, then the Bluegene/Q one. A tentative port of the Erlang VM was done on this last platform, in the prospect of being able to access a huge number of (PowerPC) cores allowing to explore the limitations met by larger simulations. Beside the CPU architecture, the port was made especially difficult by a very limited POSIX compliance (ex: regarding fork/execvp operations onto OS processes) and by the lack of a proper TCP/IP stack. A workaround was to switch the Erlang native carrier to a MPI-based one, yet the task was complex and these supercomputers were becoming increasingly superseded by HPC clusters anyway.

This finally explains why our current simulations are to run preferably on such cluster architectures when needing significant

[5]OTP stands for *Open Telecom Platform* and actually offers abstractions such as supervision trees in order to develop in any kind of fault-tolerant server application.

hardware resources. In the case of ACME, simulations do not demand a lot in terms of processing power, memory or network usage (the models involved remain fairly lightweight, and the number of their instances remains quite limited since, by nature, electrical self-consumption is to happen within local communities comprising only up of few hundred households).

If a single ACME simulation surely does not require to be distributed, as mentioned in section 4.3 their exploitation demands many simulation runs, explaining why a cluster is a tool of choice for this use case as well, and why specific needs arise at this level.

The development environment could then be quite common (x86-64 running GNU/Linux), yet in line with these final execution targets[6].

Overall, despite a certain distance between most FP languages and the HPC domain, harnessing these larger hardware resources - at least in Erlang - was not only possible but also relatively straightforward. This offered a reasonable probability that, almost regardless of their resource requirements, the functional programs discussed next could find sufficient runtime options whenever needed.

5.3 A Focus on the Myriad Layer and its Metaprogramming Facilities

The next level of the ACME software stack leaves general Erlang to enter our own set of conventions and specializations.

Myriad[12] is a generic-purpose, open-source layer whose role is to complement Erlang with the extra base functionalities to Erlang that we found useful. While not being specifically designed for Sim-Diasca, it offered a stable basis to it.

There is little interest in listing in this article the services offered by the Myriad layer, as most of them are rather mundane (merely higher-level versions of basic operations covered by the Erlang standard library, like file management, data formats, data structures, language integration, text transformations, math primitives, unit management, etc.). Yet, from a FP point of view, probably that the most interesting part of Myriad relates to its support of the built-in way of performing metaprogramming in Erlang, which is illustrated by figure 7 and is detailed here.

As mentioned, an Erlang program is nothing but a set of interacting processes. When an Erlang process is spawned, it is told which function of which module it shall execute with which arguments, and this process will simply evaluate that function (in parallel of all other processes), and terminate once done. In practice, most processes will evaluate recursive functions during which messages will be sent to other processes and be received from them.

For such a program to be built, the standard, default path is to start from a source code defining a set of modules, each defining a set of functions, some of which being exported. After various parsing and preprocessing steps, a module being compiled will end up in AST (*Abstract Syntax Tree*) form, which is an in-memory representation of its code as a data structure (i.e. an Erlang term), whose grammar covers all the elements of the language (functions, types, clauses, expressions, guards, patterns, etc. - all this according to the so-called *Abstract Format*[2]).

[6]Precisely EDF has standardized for long their hardware and software platforms with, respectively, Calibre computers running Scibian[8], an in-house GNU/Linux distribution deriving from Debian, specialized for scientific computing and natively compliant with the rest of the internal IT ecosystem.

Such a form defines completely the code of that module (and also its type specifications), and is used as the input of a multi-pass compiler which, through an intermediary, simpler language (Core Erlang), generates either assembly code for the BEAM virtual machine (as bytecodes) or, in some cases, native code (through HiPe).

So the build is to result into such per-module generated bytecodes, which are stored as BEAM files meant to be interpreted[7] at runtime by the Erlang virtual machine.

This standard compilation path is of course fully transparent to the user, and has for advantage to offer a FP paradigm which remains extremely tractable for the developer while being very effective to address concurrency and achieve seamless distribution.

As shown in figure 7, alternatively to the usual compilation path, a metaprogramming one can be gone through. Its principle is to generate from the sources the same AST as before, yet then to allow user-provided code to transform it arbitrarily, before inserting the resulting new AST in the next stages of the pipeline (linter then compiler). In Erlang parlance, such arbitrary metaprogramming code is designated as a *parse transform*, and of course is implemented in Erlang as well.

Myriad, through its AST support, can fully traverse the Erlang grammar[8] and provides a generic way of defining one's transformations, akin to a visitor pattern in which a functor could intercept language elements of interest and possibly alter them as wanted.

If such a generic service is primarily used in the above layers of the Sim-Diasca based simulations, it found also some use directly at the Myriad level, in order:

- to introduce a few, simple pseudo-builtin polymorphic types, such as a maybe-type or an associative table datatype then translated to one of our actual implementations of interest, offering various trade-offs
- to enable conditional code injection, the idea being to provide compile-time test constructs (akin, in C parlance, to `if` or `switch`) in order to select, based on tokens defined by the user, which code shall be applied[9]; this is useful, for example, to enable model-level assertions (checking pre- / post-conditions) depending on a targeted execution context
- to support the efficient sharing of mostly immutable datastructures between processes, by compiling in-memory, and possibly only at runtime from third-party data, code returning the elements that shall be shared; generating data-as-code favors scalability insofar as any number of processes can then readily access to these elements without having to duplicate these terms in their respective generational heap

Myriad also proposes its own build system (based on GNU make, yet layered as well, and comprising adequate parallel, automatic rules) instead of relying on the de facto standard rebar3[7], notably as we needed a more complex, flexible multi-stage build procedure to accommodate our use of metaprogramming.

[7]Starting from Erlang/OTP 24, a JIT (*Just-In-Time*) compiler has been introduced in order to further improve the execution efficiency.

[8]As a consequence, Myriad could metaprogram itself; however we saw no interest in introducing a corresponding bootstrap phase.

[9]As these tokens may be seen as compile-time variables, other control structures such as loops/recursion ones could be added in order to reach a sufficient expressiveness, perhaps Turing-complete, that would allow any given user program to be executed at two levels in turn, a meta one at build time and an operational one at runtime.

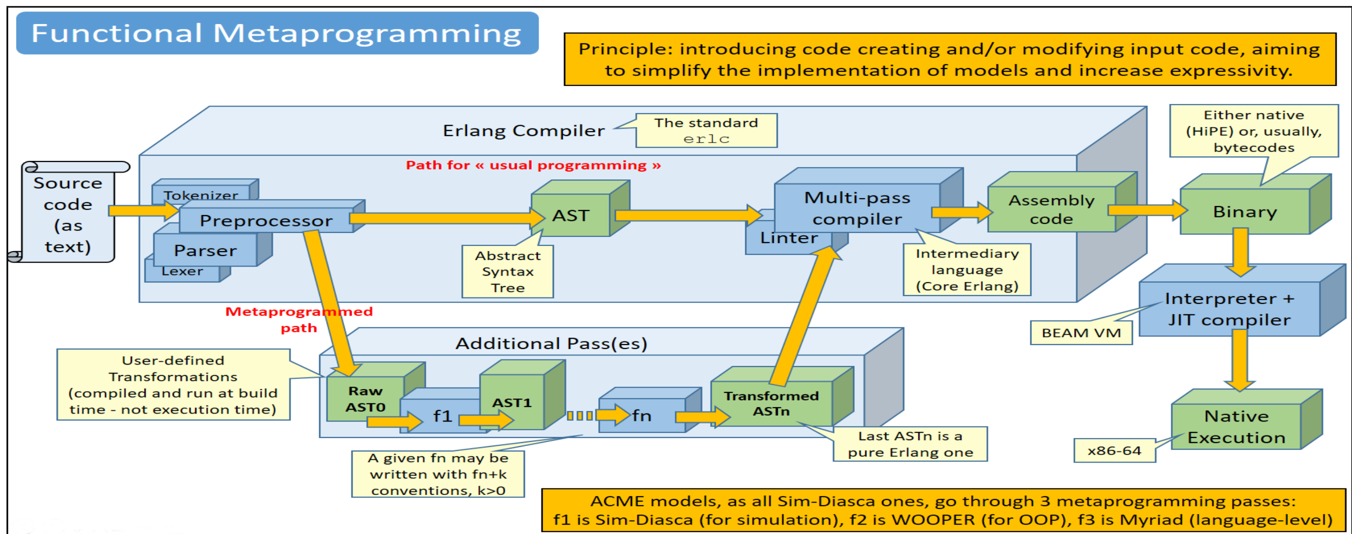

Figure 7: Use of metaprogramming implicitly done by ACME

5.4 The WOOPER Layer: a Functional Addition of an Object-Oriented Paradigm

As mentioned in section 5.2.1, Erlang offers a strong, concurrency-oriented FP paradigm, yet, as shown in section 4.1, the natural mode of expression for our models is mostly imperative (domain experts describe them as sequences of operations to be performed) and, above all, object-oriented (based on factored behaviors and states that can be collectively managed, yet still be specialised as needed).

The purpose of WOOPER[14] is to bridge this semantic gap, by augmenting Erlang with an OOP (*Object-Oriented Programming*) trait, as described by the mapping synthesised in table 1.

This WOOPER layer is based on the Myriad one, and has been similarly released as open-source software.

WOOPER predated Myriad's support for metaprogramming; as a result it used to rely exclusively on preprocessor directives, leading to limitations that the WOOPER parse transform finally removed; notably now the class developer can define any number of constructors (of any arity and any number of clauses), an optional destructor, and the various types of methods (member ones - namely oneways and requests- and static ones) are detected and to some extent checked regarding their implementation and their type specification.

Internally, a WOOPER instance (akin to a UML active object) is an Erlang process that, once constructed, loops indefinitely over a tail-recursive function whose sole parameter is the state of this instance (including a few metadata), which is implemented as an associative table storing the instance attributes.

This instance waits for incoming (Erlang) messages that directly map to as many inbound method calls, which are either oneways (one-shot, fire and forget calls) or requests (synchronous calls, returning a result to the caller process).

A class-level precomputed virtual table allows to select the relevant implementation for each of the exposed methods, according to the inheritance graph of that class.

WOOPER concept	Corresponding mapping to Erlang
Class definition	Module
Active instance	Process
Active instance reference	Process identifier (PID)
Passive instance	Opaque term
New operators	WOOPER-generated functions branching to user-defined constructors
Delete operator	WOOPER-generated function branching to any user-defined destructor
Member method definition	Module function respecting request or oneway conventions
Member method invocation	Sending of an appropriate inter-process message
Method look-up	Class-specific, inheritance-aware, virtual table
Class (static) method	Module function respecting conventions
Instance state	Set of arbitrarily-typed attributes

Table 1: Mapping from WOOPER to Erlang

Such a scheme accounts transparently for polymorphism (a WOOPER instance being designated by the PID of its hosting Erlang process, as an opaque reference, and resolving by itself the triggered methods according to its actual class) and for multiple inheritance (the per-class virtual table being preprocessed and then shared, without duplication, between its actual, direct instances); on such basis, strict encapsulation and introspection can also easily be achieved.

If a few third-party elements are confused because of the metaprogramming involved (like the Language Server Protocol or rebar3-based builds), the current implementation is to preserve most if not all features of Erlang (ex: static checking, hot code upgrade, JIT compilation), as the resulting bytecodes are pure Erlang BEAM ones.

On a side note, the metaprogramming done is so transparent and straightforward that even the code in charge of the metaprogramming is itself metaprogrammed (as, when compiled, the WOOPER parse transform is itself metaprogrammed by the Myriad parse transform). To anticipate a bit, as hinted by figure 7, further in the software stack a Sim-Diasca parse transform is in turn introduced, resulting in most modules of the ACME simulator (notably its models) going through three different passes of metaprogramming.

As a result, if Myriad-based code is mostly pure Erlang, WOOPER corresponds to a dialect that is, at least superficially, drifting further apart from the base language. Nevertheless, in practice WOOPER is just a very thin layer on top of Erlang, just augmenting it with the extra trait of OOP.

Another view on this is that WOOPER offers, thanks to Erlang, a way of defining in FP an object-oriented multi-agent system possibly of unprecedented scalability; however, as seen in section 5.6, this remains still quite far from being a simulation.

5.5 The Traces Layer, for an in-depth Monitoring of Concurrent Agents

In the context of ACME, many agents interact concurrently (ex: askers and bidders interacting through their smart contracts with the marketplace), and very soon the superposition of their behaviors cannot be anticipated (otherwise no simulation would be necessary at the first place).

The open source Traces[13] layer offers an infrastructure based on WOOPER to allow for the emission, collection and supervision of detailed parallel and distributed traces (applicative logs).

In practice, Traces introduces a `TraceEmitter` abstract mother class from which all agents able to log their state and behavior are to inherit. A dedicated parser allows to use the (optional) LogMX supervision tool. Hundred of thousands traces can then be browsed live or post-mortem, and be filtered according to several metadata (timestamps, hosts, topics, severities, emitters, etc.).

Since such traces are invaluable for many uses yet quite demanding in terms of processing resources, the least critical ones and/or those not pertaining to a set of topics of interest can be easily silenced (incurring then, thanks to metaprogramming, neither source-level code change nor any runtime overhead).

This service offers little FP challenges and thus will not be detailed further in this article. It has nevertheless proven to be of paramount importance in order to properly develop models and troubleshoot larger systems.

5.6 The Sim-Diasca Layer, to perform the actual Simulations of Complex Systems

5.6.1 Purpose. The ACME simulator is built on top of the Sim-Diasca simulation engine, a generic platform designed for the simulation of all kinds of large-scale, discrete-time, complex systems.

Most aspects of the engine will be only synthesised here, as they fall outside of the scope and size limit of this article. One may refer to the Sim-Diasca general-purpose presentation available in [30] for a more complete overview thereof.

At the heart of most simulation engines lies the management of virtual time. Most of the complex systems are modelled in discrete time, either as this matches their nature (ex: IT systems) or because, at the scale of interest, the actual physical phenomena can be safely discretized time-wise (for instance, the numerical solving of the differential equations involved in the computation of electrical load flows pertains to a different class of simulations, in answer to other use cases).

Being finely disaggregated, most complex systems comprise a large number of elements. This is especially the case for utilities, whose industrial projects lead directly to higher volumetries like the ones mentioned in section 2 (tens of thousand tasks for unit maintenance planning, dozens of millions of smart meters, etc.).

The scalability challenges underwent by the target system impact similarly their simulation, for which a sequential computation shall be ruled out: for larger systems, evaluating one model[10] instance at a time would result in intractable simulation durations. A more relevant approach is thus to opt for discrete-time *parallel* simulations.

These more advanced techniques encompass two categories of engines: synchronous, time-stepped ones (simpler but offering often lesser speedup) or asynchronous ones. This last category further ramifies into two main classes of algorithms, conservative ones (which require model-level look-ahead, deadlock detection and avoidance mechanisms) and optimistic ones (requiring complex, potentially frequent, distributed simulation rollbacks). These topics have been active areas of research for decades, and resulted in various solutions, exhibiting different trade-offs.

In the case of Sim-Diasca, the core of the management of virtual time is synchronous (enabling emulation-based use cases as discussed in section 5.2.2), with two specificities: the incorporation of asynchronous traits and the use of special simulation timestamps.

The first specificity allows a simulation case to define a fundamental, overall evaluation frequency (possibly relatively fine; for instance 50Hz for metering systems), and to have the model instances be then arbitrarily scheduled in the limits of this granularity. Models express their timings in terms as absolute durations that are adequately translated and managed by the engine, in charge of scheduling their spontaneous and triggered behaviours, based on the exchanges of so-called actor[11] messages and a hierarchy of time managers. The main feature of the engine at this level is to perform an efficient scheduling, similar to the asynchronous approaches, by jumping automatically over arbitrarily long periods without any possible activity of actors, and to ensure that all actors than can be scheduled are evaluated fully in parallel.

The second specificity relies on the use of a simulation timestamp in the form of a `(T,D)` pair, where `T` is a simulation tick (in virtual time) and `D` corresponds to a *diasca*, a concept that we introduced as a logical moment of null duration, transparently managed by

[10] A *model* shall be understood here as a simplified, abstract representation of an element of the target system of interest. Like a model of a battery, or of an electric vehicle, a marketplace, a smart contract, etc.

[11] *Actor* being used here as a shorthand for *model instance*.

the engine in order to sort out causality and the other properties discussed next.

Indeed the simulations that we target are to meet following three properties, relatively straightforward to enforce in a sequential context, yet considerably more problematic to obtain efficiently in a concurrent one:

(1) the respect of causality: without a proper reordering of inter-actor messages done by the engine, on a regular basis causes would happen after their effects from the point of view of some actors[12], leading to incorrect model evaluation; Sim-Diasca performs such an automatic, transparent reordering, moreover in a massively parallel manner
(2) a total reproducibility (a simulation run twice should follow exactly the same trajectory and yield the same results), which is not granted by default either, not only because of parallelism and distribution, but also because models may be stochastic, i.e. may be at least partly ruled by probability density functions
(3) some form of ergodicity, so that:
    (a) all possible outcomes according to the models can actually occur in the simulations
    (b) their probability of showing up in the simulations is close to the one that could be determined a priori from the models

By providing adequate constructs (process isolation and scheduling, higher-level asynchronous messaging, seamless distribution) and safety nets (ex: automatic memory management, immutability and the limitation of side-effects), a FP solution like Erlang has been instrumental to the design of such an intensely concurrent mode of operation, and resulted in a tool able to scale up significantly[13]: at the algorithmic level, a maximal parallelization of the evaluation of models could be devised, whereas at the runtime level, as mentioned in section 5.2.3, adequate computing resources could be harnessed (multicores, SMP, clusters and other HPC solutions).

We focused in this article on time management, as this is a domain that benefited a lot from FP. Yet quite many other services, which will be only lightly mentioned here due to size constraints, have also to be provided in order to properly support simulations and conduct virtual experiments such as ACME; like:

- the definition of simulation cases, including the creation of a relevant initial state - programmatically or through file-based, parallel initialisation
- the automatic deployment of the full resulting simulator (code and data) on a set of computing nodes, with no prior installation thereof except Erlang itself (optionally integrated to a cluster job manager like Slurm)
- the life-cycle management of all actors (initial or created by other actors in the course of the simulation)
- the load balancing of a simulation running across several computing hosts, including the management of placement hints in order to co-allocate model instances known to be tightly coupled
- the management of simulation results, thanks to basic or specialised probes fed with samples sent by actors and aggregated in simulation metrics (everything done concurrently as well)
- auxiliary services, such as performance tracking, data exchanger (to provide extra communication tradeoffs), dataflow support, stochastic support

An insight of this work is that thanks to overall conventions, language facilities and simulation services, the models are sheltered from most of the complexity of the simulation. Indeed, despite the several layers presented here and the requirements to fulfill, writing a model merely boils down to defining the structure of its state and then specifying its behavior (both spontaneous and triggered), through only very simple, sequential, autonomous actions: (1) operating internal state changes, (2) planning future spontaneous schedulings for oneself and (3) sending actor messages to others.

6 SOME FP-RELATED LESSONS LEARNED THROUGH PRACTICE

Over the years, in addition to ACME, a few simulators based on Sim-Diasca have been developed internally to EDF R&D on behalf of various projects, regarding smart metering infrastructures, urban planning or the organisation of complex unit maintenance.

A lesson that these experiments taught us is that, for similar studies to be done in a FP context, at least two routes could be considered and contrasted.

The first one, that can be illustrated for example by a former IFL article on a related topic[3], is directly based on high-end, comprehensive FP abstractions, involving typically strong static type system like the provided by the Haskell language, monadic stream functions and a wide range of advanced techniques in order to favor purity and better delimit by design side effects.

The second one, discussed in this article, offers two levels: first a very straightforward, pragmatic approach (like done with pure Erlang, or with our simple, protected setting in which user models are to be implemented), then a fully optional, more advanced level that enables more powerful solutions (like when making use of metaprogramming, or entering the domain of the engine itself - in which most of the complexity is deported - and the extra services that may be then considered).

The first high-end approach provides unrivaled guarantees about the properties met by simulations even before they are run. The second approach brings another kind of benefit: general simplicity, moreover at an adjustable level.

Our initial belief was that this last approach could become limiting and/or hit performance issues. The plan was to gradually enhance the weaker points of the engine on a per-need basis, possibly by going for increasingly static typing and a stricter concurrency model. Each time Erlang would have been insufficient, a more adapted language would then be considered (ex: possibly allowing models to be implemented, thanks to language bindings, in Haskell for expressiveness, or in Rust for raw performance).

However we actually never really reached such hard limits, whereas in the meantime considerable latent leeway appeared thanks to metaprogramming potentialities, algorithmic enhancements and the now operational JIT compiler.

[12]Typically, as we detailed in [30], 3 actors, each running on a different computing node, and as many actor messages are sufficient to showcase a breach of causality.
[13]Even back in 2009 we were able to evaluate, over a few nodes of clusters of that time, distributed simulations comprising each more than 1 million instances of rather heavyweight models, evaluated in parallel.

Instead we were repeatedly hit by a significantly more concrete issue, still relating to FP and that would have been especially exacerbated in the first, high-end approach: as soon as such a simulation project was proving its usefulness, ramping up the developing resources became both a necessity and a problem. Indeed, requesting from subcontractors additional workforce in terms of developers with prior FP skills proved to be a challenge. Too often, offers were either non-existing or inadequate (inexplicably expensive or not able to complement an in-house team due to distance, work practices or spoken language). Albeit this is certainly a down to earth problem, such pitfalls could be largely sufficient to preclude at least some applications of FP in an industrial context.

We found however a rather effective workaround by, in several occasions, hiring junior engineers (not necessarily developers) with no prior FP background, and training them internally regarding the software stack presented in this article. Thanks to the simplicity of the FP approach taken, they could be brought up to speed within a few weeks, and were able to contribute, often with great success, to these simulation projects. This has been the case for ACME.

7 FINAL WORD & PERSPECTIVES

In this article, we discussed the development of a simulator for a local energy marketplace and showcased the industrial use of a functional language like Erlang for the simulation for real-world systems.

We explained how, in the context of this ACME project, we translated a UML 2.0 description obtained from domain experts into a complete simulation able to take advantage of the intrinsic properties of Erlang in favor of massive parallelism and distribution.

The software layers involved (Myriad, WOOPER, Traces and Sim-Diasca) facilitating such translation for the user have been also been described, and examined in a FP perspective.

Future work may include a tool for automatic code generation for multi-agent simulations based on the model driven development methodology (through the use of formal description of the agents at the analytical level and transformation software based on WOOPER for automatic code generation), extra communication primitives allowing to further mask the underlying parallelism for model-level synchronous interactions, and newer applications of the Sim-Diasca engine to the simulation of large-scale business IT systems, in the prospect of obtaining a digital twin thereof.

REFERENCES

[1] [n.d.]. *CAP Theorem.* https://en.wikipedia.org/wiki/CAP_theorem

[2] [n.d.]. *Erlang Abstract Format.* https://erlang.org/doc/apps/erts/absform.html

[3] 2018. ACM. https://doi.org/10.1145/3310232.3310372

[4] 2021. *Dialyzer.* https://erlang.org/doc/apps/dialyzer/dialyzer_chapter.html

[5] 2021. *EDF.* https://www.edf.fr/en/the-edf-group/edf-at-a-glance

[6] 2021. *OpenMOLE.* https://openmole.org/

[7] 2021. *Rebar3.* http://rebar3.org/

[8] 2021. *Scibian.* https://scibian.org/

[9] 2021. *Tilera.* https://en.wikipedia.org/wiki/TILEPro64

[10] Rabia Aziza, Amel Borgi, Hayfa Zgaya, and Benjamin Guinhouya. 2016. Simulating Complex Systems - Complex System Theories, Their Behavioural Characteristics and Their Simulation. 298–305. https://doi.org/10.5220/0005684602980305

[11] Federico Bergenti and Agostino Poggi. 2000. Exploiting UML in the Design of Multi-agent Systems. In *Engineering Societies in the Agents World*, Andrea Omicini, Robert Tolksdorf, and Franco Zambonelli (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 106–113.

[12] Olivier Boudeville. 2021. *Ceylan-Myriad.* https://myriad.esperide.org/

[13] Olivier Boudeville. 2021. *Ceylan-Traces.* https://traces.esperide.org/

[14] Olivier Boudeville. 2021. *Ceylan-WOOPER.* https://wooper.esperide.org/

[15] CEA/EDF/INRIA. 2012. *Functional Programming for Parallel and Concurrent Applications.* http://www-hpc.cea.fr/SummerSchools2012.htm

[16] Jesús Cerquides, Gauthier Picard, and Juan Rodríguez-Aguilar. 2015. Defining a Continuous Marketplace for the Trading and Distribution of Energy in the Smart Grid. 2 (05 2015).

[17] Asmae Chakir, Mohamed Tabaa, Fouad Moutaouakkil, Hicham Medromi, Maya Julien-Salame, Abbas Dandache, and Karim Alami. 2020. Optimal energy management for a grid connected PV-battery system. *Energy Reports* 6 (2020), 218–231. https://doi.org/10.1016/j.egyr.2019.10.040 Technologies and Materials for Renewable Energy, Environment and Sustainability.

[18] Ericsson. 2021. *Erlang Type Specifications.* https://erlang.org/doc/reference_manual/typespec.html

[19] Ericsson. 2021. *Erlang/OTP official website.* http://erlang.org

[20] Guillaume Guérard, Soufian Ben Amor, and Alain Bui. 2012. A Complex System Approach for Smart Grid Analysis and Modeling, Vol. 243. 788–797. https://doi.org/10.3233/978-1-61499-105-2-788

[21] Pu Huang, Alan Scheller-wolf, and Katia Sycara. 2002. A Strategy-Proof Multiunit Double Auction Mechanism Pu Huang. (12 2002).

[22] Enrique Kremers. 2020. Intelligent local energy management through market mechanisms: Driving the German energy transition from the bottom-up. *Energy Reports* 6 (05 2020), 108–116. https://doi.org/10.1016/j.egyr.2020.03.004

[23] Alexandra Lüth, Jan Martin Zepter, Pedro Crespo del Granado, and Ruud Egging. 2018. Local electricity market designs for peer-to-peer trading: The role of battery flexibility. *Applied Energy* 229 (Nov. 2018), 1233–1243. https://doi.org/10.1016/j.apenergy.2018.08.004

[24] Esther Mengelkamp, Samrat Bose, Enrique Kremers, Jan Eberbach, Bastian Hoffmann, and Christof Weinhardt. 2018. Increasing the efficiency of local energy markets through residential demand response. *Energy Informatics* 1 (08 2018). https://doi.org/10.1186/s42162-018-0017-3

[25] Esther Mengelkamp, Johannes Gärttner, Kerstin Rock, Scott Kessler, Lawrence Orsini, and Christof Weinhardt. 2018. Designing microgrid energy markets. *Applied Energy* 210 (Jan. 2018), 870–880. https://doi.org/10.1016/j.apenergy.2017.06.054

[26] Esther Mengelkamp, Benedikt Notheisen, Carolin Beer, David Dauer, and Christof Weinhardt. 2017. A blockchain-based smart grid: towards sustainable local energy markets. *Computer Science - Research and Development* 33, 1-2 (Aug. 2017), 207–214. https://doi.org/10.1007/s00450-017-0360-9

[27] Esther Mengelkamp, Philipp Staudt, Johannes Garttner, and Christof Weinhardt. 2017. Trading on local energy markets: A comparison of market designs and bidding strategies. In *2017 14th International Conference on the European Energy Market (EEM)*. IEEE. https://doi.org/10.1109/eem.2017.7981938

[28] RELEASE Project. 2011. *A HIGH-LEVEL PARADIGM FOR RELIABLE LARGE-SCALE SERVER SOFTWARE.* http://www.release-project.eu

[29] EDF R&D. 2021. *Public Sim-Diasca project.* https://github.com/Olivier-Boudeville-EDF/Sim-Diasca

[30] EDF R&D. 2021. *Public Sim-Diasca wiki.* https://github.com/Olivier-Boudeville-EDF/Sim-Diasca/wiki

[31] EDF R&D. 2021. *Simulation Software.* https://www.edf.fr/en/the-edf-group/inventing-the-future-of-energy/r-d-global-expertise/our-offers/simulation-softwares

[32] Maria Vasconcelos, Wilhelm Cramer, Carlo Schmitt, Arvid Amthor, Stefan Jessenberger, Christian Ziegler, Andreas Armstorfer, and Florian Heringer. 2019. The PEBBLES project – enabling blockchain based transactive energy trading of energy flexibility within a regional market. https://doi.org/10.34890/591